LOADING CSV FILES INTO Q

Ramesh Subramonian

Abstract

A At a very high level, when we load data from a CSV file, we are converting data presented in ascii in row format to a set of files in binary format, one for each of the fields in the CSV file. Technically, we are creating a table of vectors but we will get to that detail later in this document.

1 Introduction

1.1 Pre-Reading Material

read `https://www.lua.org/pil/12.html`

1.2 What is a CSV file?

A CSV file is an ascii file such that:

1. records are terminated by eoln character `'\n'`
2. fields are separated by comma character
3. fields may be enclosed by a double-quote character e.g., `1,"foo",2.3`
4. a single record can span multiple lines
5. a null value can be represented either by 2 consecutive double-quote characters e.g., the following lines are identical

```
foo,,2
foo,"",2
"foo,"","2"
```

6. all rows must have the same number of fields
7. Three characters have special meanings - comma, double quote and eoln. When we want these to occur as part of the value, they must be preceded by a back-slash. So, we really have 4 special characters. For example, the value `"abc\,\\\""` has 6 characters (1) a (2) b (3) c (4) comma (5) backslash and (6) double quote

1.3 CSV Customizations

At some future point, we might want to make the following customizable

1. field separator, currently comma

2. record delimiter, currently eoln
3. field opener, currently double-quote
4. field closer, currently double-quote

1.4 Environment Variables

The following environment variables are mandatory

1. `Q_DATA_DIR` This is where binary files corresponding to vectors will be stored.
2. `Q_META_DATA_DIR` This is where meta data needed to restore state will be stored

1.5 Conventions

All binary files will

1. start with underscore
2. be followed by a sequence of alphanumeric characters
3. contain exactly 16 characters

2 Q syntax

The Q operator that loads a CSV file is as follows

```
X = load(data="foo.csv", meta_data = M, global_meta = M2 }
```

We now explain M and M2. M2 has information that is pertinent to the processing of the entire file. In addition to the information in Section 1.3, it has

1. is_hdr = true or false, which tells us whether to ignore the first line or not

M is a table that has N values, one for each of the N fields that we expect to find in foo.csv

What does M look like?

```
M = { { name = "", field_type = "" },
{ name = "f1", null = "true", type = "int32_t" },
{ name = "f2", null = "false", type = "integer" },
{ name = "f3", type = "number" },
{ name = "f4", size = 16 },
```

```
{ name = "f5", type = "varchar", dict = "D1", is_dict = false },
{ name = "f6", type = "varchar", dict = "D2", is_dict = true, add=true},
{ name = "f7", type = "varchar", dict = "D3", is_dict = true, add=false},
    }
```

Let's go through these lines one at a time. The table has 6 fields.

1. The first line says that the first field should be ignored - hence we do not need to provide a name or a field type
2. Line 2 for field f1 says that we expect all values of the second field to be valid values for a variable of type `int32_t` and that the resulting vector is of type `int32_t`
3. Line 3 for field f2 says that we expect all values of the third field to be valid values for a variable of type `int64_t` and that the resulting vector is of type `int8_t, int16_t, int32_t, int64_t`. The smallest field that can accomodate the values will be used. In other words, if all values are between -32768 and +32767, then we will use `int16_t` Not for version 1.
4. Line 4 for field f3 says that we expect all values of the third field to be valid values for a variable of type double and that the resulting vector is of type `int8_t, int16_t, int32_t, int64_t, float, double`. If all values can be represented by an integer, the smallest integer field will be used. Else, float will be used if it causes no loss; else, double. Not for version 1.
5. Line 5 for field f4 says that this is a constant length string. The size includes the null character used to terminate the string. So, 0123456789ABCDE is a valid string for length 16 but 0123456789ABCDEDF is not. If the input value is less than 15 characters, it is right-padded with null characters so that the total length of the field (as written to the binary file) is 16 characters.
6. In order to explain lines with "dict", we need to introduce the concept of a dictionary, Section 4

3 Null values

We now explain the **null** in the meta data of the previous example.

Let us say that a vector x has null values and is stored in file `_x`. Then, we will have a nn file called `_nn_x` which will tell us which values of x are null and which are not.

What does the nn file look like? Our initial idea was to use an 8-bit integer to represent 1 and 0. We then decided to use bits instead. However, there is an important subtlety regarding the size of the nn file to be aware of.

If we had used a byte to record whether null or not, then if x had 1000 values and was of type `int32_t`, then `_x` would be 4000 bytes in length and `_nn_x` would be 1000 bytes in length. When the nn file is recorded as bits and not bytes, then we need to **round up** to the next multiple of 64. Since 1024 is the smallest value greater than equal to 1000 which is a multiple of 64, then `_nn_x` would have length 1024 bits = 128 bytes

What happens when the user specifies that a vector has no null values but we encounter a null value while reading the data file? Abort the entire operation.

What happens when the user specifies that a vector has null values but we do **not** encounter a null value while reading the data? This is legitimate. However, we delete the nn file created before creating the vector.

What happens if the user does not specify whether a vector has null values? In this case, we assume that `null = "true"`

For later: we will need these macros

1. `set_bit(N, i, 1)`
2. `set_bit(N, i, 0)`
3. `get_bit(N, i)`

4 Dictionaries

Q does not (for now) support variable length character strings as a data type. So, when we read a field that contains a character string, we convert it into an integer. A dictionary is the data structure that allows us to deterministically map from strings to integers on the input and vice versa on the output.

During loading, there are two main cases to consider

1. `is_dict=false` no dictionary exists for this field and one must be created. If D1 is such a dictionary, then no other field in this load statement can use D1. If a dictionary called D1 exists, then this operation aborts.
2. `is_dict=true` a dictionary exists for this field and must be used. In this case, we have two sub-cases. If you come across a string which does not exist in the dictionary,
   (a) `add=true` can you add to the dictionary.
   (b) `add=false` can you **not** add to the dictionary.

   If D2 is such a dictionary, then it can be used by other field so long as they use it in a similar manner. If a dictonary D2 does not exist, the operation aborts. If `add=false` and you come across a string that is not in D2, the operation aborts.

A few important implementation notes:

1. Use Lua to parse the file up to the point where you have identified the string that corresponds to a particular cell in the table represented by the CSV file. At that time, hand off to C to convert this into a valid C type, as in Section 1. As an example, the C function takes the string
   (a) `123456` and creates the 4 bytes corresponding to an `int32_t` representation of 123456.
   (b) `abc` and creates the 8 bytes `abc00000` where 0 represents the null character and we assume that the user wanted a constant length string field of length 8.
2. When the system shuts down, it is important that the dictionaries are persisted to disk because they will need to be restored when the
3. Our current approach relies on the dictionaries to be Lua tables i.e., we are relying on Lua's fairly efficient implementataion of hash tables. However, if the number and size of these fields becomes very large, then we will need to explore other alternatives e.g. other key value stores like Redis, LevelDB, ...

5 Custom Types

Types are registered globally with the following information.

1. short code
2. C equivalent
3. description (optional)
4. input text converter. This is a C function which has the signature shown below. The function returns 0 if the conversion was successful and a negative number otherwise Assume that the short code is `B` and that the C equivalent is `int8_t`. In that case, the converter is

   ```
   int txt_to_c( const char *in, int8_t *ptr_out)
   ```

5. output text converter. This is a C function which has the signature shown below. The function returns 0 if the conversion was successful and a negative number otherwise Assume that the short code is `B` and that the C equivalent is `int8_t`. In that case, the converter is

   ```
   int c_to_txt(int8_t in, char *out, ssize_t sz_out)
   ```

The 6 types that come with the base Q installation are in Table 1

Short Code	C Equivalent	Description	Input Converter	output Converter
c	`int8_t`		strtod	fprintf, %d
s	`int16_t`		strtod	fprintf, %d
i	`int32_t`		strtod	fprintf, %d
l	`int64_t`		strtoll	fprintf, %lld
f	`float`		strtof	fprintf, %f
d	`double`		strtold	fprintf, %lf
t	struct tm	timestamp	strptime+	strftime

Table 1: Pre-built types